Chapter 2
Deep Learning Basics

This chapter describes several commonly used algorithms and basic concepts related to deep learning.

2.1 Regression Algorithms

Regression algorithms typically use a series of properties to predict a value, and the predicted values are continuous. For example, the price of a house is predicted based on the house's features, such as its location and number of bedrooms, or weather forecasts are predicted according to the previous week's temperatures and weather maps. Using the first example, if the actual house price is CNY 5 million, and the value predicted through regression analysis is CNY 4.99 million, the regression analysis is considered accurate. For machine learning problems, regression analysis includes linear regression, polynomial regression, logistic regression, and others. This section focuses on linear regression algorithms due to their simplicity, offering an ideal starting point for those wanting to quickly understand the basics of deep learning. Section 2.3 builds on this basic understanding by elaborating on logistic regression.

Let's start by looking at a linear regression problem with only one variable. Table 2.1 shows the given data, also called a dataset.

Each group of data is denoted as $(x^{(i)}, y^{(i)})$, and there are m groups of data in total. The goal is to obtain a model through which a value of y is predicted based on a newly given value of x. For linear regression, the model is a linear function, and the formula is as follows:

$$h(x^{(i)}) = w_0 + w_1 x^{(i)} \tag{2.1}$$

where w_0 and w_1 represent parameters that need to be obtained through training.

L. Chen, *Deep Learning and Practice with MindSpore*, Cognitive Intelligence and Robotics, https://doi.org/10.1007/978-981-16-2233-5_2

Table 2.1 Given data

x	– 12	– 4	1	10	20	29	43	60	…
y	0	2	3	5	8	7	10	15	…

This model is also called hypothesis. Linear regression aims to obtain an optimal group of w_0 and w_1, so that the hypothesis is close to the dataset shown in Table 2.1. In this way, we hope that the w_0 and w_1 obtained through training can fit the given dataset as closely as possible, as shown in Fig. 2.1.

So how do we obtain the optimal w_0 and w_1? We can achieve this by converting the training goal into minimization of the following function:

$$J(\boldsymbol{w}) = \frac{1}{2m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2 \tag{2.2}$$

A function that needs to be minimized is called a loss function, and more than one may exist. The one given above is called the mean square error function, which is typically used to solve regression problems. For classification problems, the cross-entropy loss function is typically used. An example of this is:

$$J(\boldsymbol{w}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log h(x^{(i)}) + (1 - y^{(i)})\log(1 - h(x^{(i)})) \tag{2.3}$$

The final optimization goal is to minimize the error between the predicted value $h(x^{(i)})$ and the actual label $y^{(i)}$ in the training data.

For the sake of simplicity, let's assume that the input data x has only one property, although in practical applications it may include n properties ($n \geq 1$). In this case, $n + 1$ parameters $\boldsymbol{w}$ need to be obtained through training, $\boldsymbol{w} = [w_0, w_1, \ldots, w_n]$, where w_0 is a bias, and w_i ($i = 1, 2, \ldots, n$) is a weight of the ith property. In summary, the regression problem can be expressed as follows:

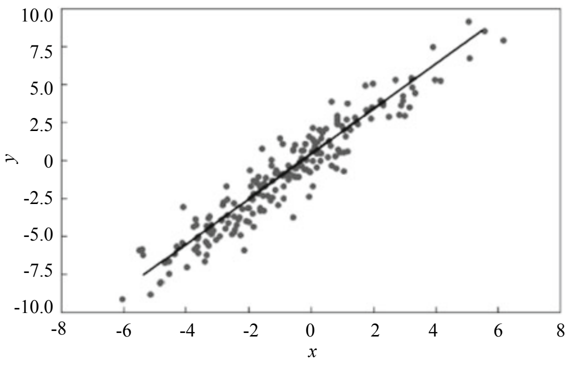


Fig. 2.1 Linear regression model

(1) Dataset

$$\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}, \boldsymbol{x} = [x_1, x_2, \cdots, x_n] \tag{2.4}$$

(2) Parameter

$$\boldsymbol{w} = [w_0, w_1, \cdots, w_n] \tag{2.5}$$

(3) Hypothesis

$$h(x^{(i)}) = w_0 + \sum_{j=1}^{n} w_j x_j^{(i)} \tag{2.6}$$

(4) Loss function

$$J(\boldsymbol{w}) = \frac{1}{2m} \sum_{i=1}^{m} (h(x^{(i)}) - y^{(i)})^2 \tag{2.7}$$

For brevity, the dataset is expressed as an $m \times (n + 1)$ matrix $\boldsymbol{X}$, where the first element of each row is always 1, which is followed by n properties of $x^{(i)}$, that is:

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \tag{2.8}$$

In addition, the target value is also denoted as a vector form $\boldsymbol{y} = (y^{(1)}; y^{(2)}; \ldots; y^{(m)})$, meaning that the linear regression problem can be expressed as follows:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \frac{1}{2m} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^{\mathrm{T}} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) \tag{2.9}$$

In order to find an extremum $\boldsymbol{w}^*$, we can calculate the gradient of $\boldsymbol{w}$ to obtain:

$$\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{1}{m} \boldsymbol{X}^{\mathrm{T}} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \tag{2.10}$$

When $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ is a full-rank matrix or a positive-definite matrix, let the gradient $\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = 0$ in order to obtain:

$$\boldsymbol{w}^* = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1} \boldsymbol{X}^{\mathrm{T}} \boldsymbol{y} \tag{2.11}$$

where $(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}$ is an inverse matrix of the matrix $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$.

After obtaining $\boldsymbol{w}^*$, for any sample $\boldsymbol{x}$, let $\widehat{\boldsymbol{x}} = [1, \boldsymbol{x}]$, so that we can predict a regression value of $\boldsymbol{x}$ as follows:

$$h(\boldsymbol{x}) = \widehat{\boldsymbol{x}}^{\mathrm{T}}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y} \tag{2.12}$$

However, in practical applications, $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ is unlikely to fully meet the conditions of the full-rank matrix or the positive-definite matrix. For example, the quantity of properties in some tasks might be large, potentially even large enough to exceed the quantity m of samples, that is, $n \geq m$. As a result, the quantity of columns in the sample matrix $\boldsymbol{X}$ would be much larger than the quantity of rows. In this case, $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ is unable to meet the conditions of the full-rank matrix.

In the linear regression problem, let $\frac{\partial J(\boldsymbol{w})}{\partial \boldsymbol{w}} = 0$, so that we can obtain an analytical solution of the optimal parameter $\boldsymbol{w}^*$. However, for models or loss functions that are more complex, there is usually no analytical solution. Section 2.2 will describe gradient descent, an algorithm that has found wide use in the field of machine learning. By minimizing the loss function, this algorithm obtains, through training, the parameter $\boldsymbol{w}^*$ that needs to be calculated.

2.2 Gradient Descent

Gradient descent is a first-order optimization algorithm used to find the local minimum of a loss function (or objective function). Using this algorithm, also called steepest descent, we need to move a distance proportional to the negative of the gradient (or approximate gradient) corresponding to the current point of the function in order to implement iterative search. The converse of this is called gradient ascent, whereby if we move a distance proportional to the positive of the gradient, we will approach the local maximum point of the function. This section focuses exclusively on gradient descent.

Gradient descent is based on the following observation: If a real function $J(\boldsymbol{w})$ is differentiable and defined at w, then $J(\boldsymbol{w})$ decreases fastest at the point $\boldsymbol{w}$ along the direction $-\nabla J(\boldsymbol{w})$, which is opposite to the gradient. As shown in Fig. 2.2, if we

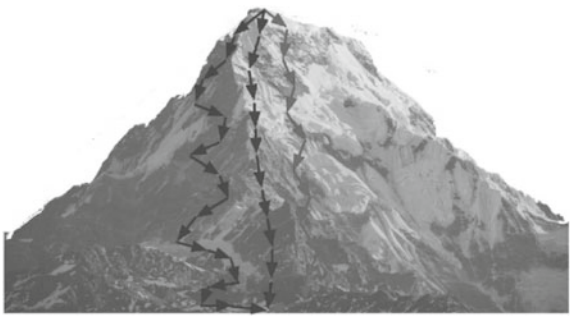


Fig. 2.2 Gradient over ridges

follow the middle route, we would travel a much shorter distance than if we traveled along either the left or right route.

Based on this, the concept of gradient descent includes the following parts:

(1) Selecting an initial point w_0.
(2) Gradually updating the parameter in the negative direction of the gradient, $w_t = w_{t-1} - \alpha \nabla J(w_{t-1})$ until convergence.

Here, $\alpha > 0$. Also known as learning rate, α is step size whose value can be set before training, or it can be adjusted according to the training situation. Based on the definition of gradient, if the value α is small enough, $J(w_t) \leq J(w_{t-1})$. If an appropriate step size is given, starting from w_0, smaller loss functions $J(w_0) \geq J(w_1) \geq J(w_2) \geq \ldots$ are obtained gradually. In theory, the sequence w_t will gradually converge to the minimum of the loss function, as shown in Fig. 2.3.

Assume that the loss function J is defined on a plane and looks like a bowl. The elliptical curves represent contour lines; that is, the function J is a curve formed by a set of constants, where the value becomes smaller as it approaches the center. If an initial point w_0 is selected arbitrarily, the arrows point to the negative direction of the gradient (the gradient direction is perpendicular to the contour line of the point). In this case, the parameter $\boldsymbol{w}$ is gradually updated along the descending direction of the gradient until it reaches the bottom of the bowl, that is, it reaches the minimum point of the function J.

Because we can convert the problem of solving the linear regression, mentioned in Sect. 2.1, into one of minimizing the loss function $J(\boldsymbol{w})$, we can use gradient descent here. For a given dataset $\{(x^{(i)}, y^{(i)})\}$, the parameter $\boldsymbol{w}$ is defined according to the number of properties of $\boldsymbol{x}$. The linear equation $h(x^{(\mathrm{i})})$ and the loss function are as follows:

$$J(\boldsymbol{w}) = \frac{1}{2m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2 \tag{2.13}$$

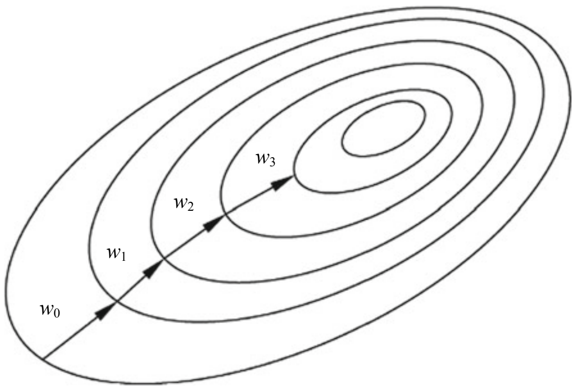


Fig. 2.3 Illustration of gradient descent

First, we need to randomly select an initial point w_0 and an appropriate step size α. The gradient calculation formula is as follows:

$$\nabla J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})\hat{x}^{(i)} \tag{2.14}$$

where $\hat{x}^{(i)} = [1, x^{(i)}]$. A dimension of a constant 1 is added to the original property vector to update the bias w_0.

The parameter w_t can be updated step by step based on the gradient $\nabla J(\boldsymbol{w})$, and the optimal parameter value can be obtained by using $w_t = w_{t-1} - \alpha\nabla J(w_{t-1})$. In addition to being used in linear regression, gradient descent can also be used for minimizing the loss function to solve many problems in machine learning.

This versatile algorithm is also called batch gradient descent, where "batch" means that the total quantity m of training samples is used. In practical problems, the quantity of samples may be large (e.g., the quantity of students in a school, clients in a bank, and files on a hard disk). The quantity of parameters may also large, especially for complex learning models such as the DNN. Using all data samples in each gradient calculation will produce large results or even make the calculations impossible. The batch gradient descent algorithm can be thought of as a random process, in which only one point is randomly selected at a time, which is expected to be roughly similar to the sum of all points. This means that the gradient of a single point, called stochastic gradient, can be used instead of the average gradient, and the overall gradient can be regarded as an expected value of the stochastic gradient. An iterative algorithm for a linear programming problem based on stochastic gradient descent involves the following formula:

$$w_t = w_{t-1} - \alpha(h(x^{(i)}) - y^{(i)})\hat{x}^{(i)} \tag{2.15}$$

where $x^{(i)}$ is the sample randomly selected from m data samples during the tth iteration.

Because only one sample is used for each update without traversing all datasets, the speed of iteration is relatively fast. However, the gradient direction selected each time may be not optimal due to a deviation of random sampling, meaning that the quantity of iterations and the minimum value of convergence may also not be optimal.

In practical applications, mini-batch gradient descent—a trade-off between batch gradient descent and stochastic gradient descent—is a more popular algorithm. A small batch of samples with a sample quantity of b ($b < m$) are randomly selected each time, speeding up calculation of the entire batch. In addition, using a small batch in calculating the gradient direction yields greater accuracy than the stochastic gradient direction, which is based on only one sample. Mini-batch gradient descent is shown in Algorithm 2.1.

Algorithm 2.1 Mini-batch gradient descent

```
Input: Dataset $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, step size $\alpha$, mini-batch training sample size $b$, and iteration quantity $T$
Output: Converged parameter $w_T$
(1) Initialize the parameter $w_0$
(2) for $t \in \{1, 2, ..., T\}$.
(3)     Select $b$ samples uniformly at random from $m$ samples
(4)     Calculate the gradient and update the parameter
```

$$w_t \leftarrow w_{t-1} - \frac{\alpha}{b} \sum_{i \in m_b} \partial_w J_i(w) \tag{2.16}$$

Algorithm 2.1 summarizes the main process of mini-batch gradient descent, where m_b is an index set of b samples randomly selected from m samples, and $J_i(\boldsymbol{w})$ is a loss function on the ith sample. We will describe selection of a step size, conditions for convergence, and other relevant aspects in subsequent sections.

2.3 Classification Algorithms

Unlike a regression problem, the output of a classification problem is a discrete value rather than a continuous value, that is, the category of the sample. Classification problems are widely used in real-world applications, for example, to perform handwritten digit recognition, email spam classification, and facial recognition, and even to distinguish cats and dogs in images. Such problems include binary classification ("yes" or "no") and multi-class classification (where one of multiple categories needs to be determined). All multi-class classification problems can be converted into multiple binary classification problems. For example, when animals are classified, each animal can be systematically determined using "yes" or "no" in order to achieve the goal of multi-class classification. This section focuses on the binary classification problem.

Consider the following question: Can we solve a classification problem by using the same method we use to solve a regression problem? The answer is yes. The main difference between classification and regression problems is that a series of discrete values, rather than a straight line or curve, is to be fit. The logistic regression mentioned in Sect. 2.1 is ideal for the linear binary classification problem.

Logistic regression is based on the logistic function (also known as a log probabilistic function) shown in Fig. 2.4, that is:

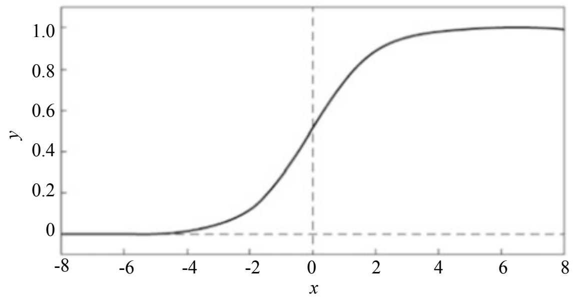


Fig. 2.4 Logistic function $\text{sigmoid}(x) = \frac{1}{1+\mathrm{e}^{-x}}$

$$\text{sigmoid}(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{2.17}$$

where $\mathrm{e}^{(\cdot)}$ is an exponential function.

Any given input value x is mapped by the logistic function to a value in the range 0–1. The output value is closer to 1 if $x > 0$, and closer to 0 if $x < 0$. When $x = 0$, it indicates an inflection point with the maximum slope value. The output value of the logistic function is usually regarded as the probability that the output is 1. The logistic function is a monotonically differentiable function, which is an important property that allows gradient descent to be used for model training.

For a given sample property x, assume that the label $y \in \{0, 1\}$, and the predicted value $z(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\widehat{\boldsymbol{x}}$, where $\widehat{\boldsymbol{x}} = [1, \boldsymbol{x}]$. For classification problems, if $z(\boldsymbol{x}) > 0$, the predicted value is 1, whereas if $z(\boldsymbol{x}) < 0$, the predicted value is 0. In this way, the predicted value $z(\boldsymbol{x})$ can be substituted into the logistic function to obtain:

$$h(\boldsymbol{x}) = \frac{1}{1 + \mathrm{e}^{-\boldsymbol{w}^{\mathrm{T}}\widehat{\boldsymbol{x}}}} \tag{2.18}$$

From the perspective of probability, the classification probability can be expressed as $P(y = 1|\boldsymbol{x}, \boldsymbol{w}) = \frac{1}{1+\mathrm{e}^{-\boldsymbol{w}^{\mathrm{T}}\widehat{\boldsymbol{x}}}}$. Similarly, $P(y = 0|\boldsymbol{x}, \boldsymbol{w}) = \frac{1}{1+\mathrm{e}^{\boldsymbol{w}^{\mathrm{T}}\widehat{\boldsymbol{x}}}}$.

For a given dataset $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, in the linear regression problem, the optimization goal is to minimize the mean square error. However, in the logistic regression model, the optimization goal is to maximize the likelihood function L. The calculation formula for L is as follows:

$$L(\boldsymbol{w}) = \prod_{i=1}^{n} P(y^{(i)}|x^{(i)}, \boldsymbol{w}) = \prod_{i=1}^{n} (h(x^{(i)}))^{y^{(i)}}(1 - h(x^{(i)}))^{1-y^{(i)}} \tag{2.19}$$

If the quantity of samples is too large, numeric overflow may easily occur due to the characteristics of computer floating-point numbers. Because the form of consecutive addition is easier than consecutive multiplication, in terms of gradient calculation, we usually take the logarithm of multiple consecutive multiplication terms. Specifically, we can use the log-likelihood function shown in the following formula:

$$l(\boldsymbol{w}) = \log L(\boldsymbol{w}) = \sum_{i=1}^{n} y^{(i)} \log(h(x^{(i)})) + (1 - y^{(i)}) \log(1 - h(x^{(i)})) \tag{2.20}$$

Now we can use the gradient descent described in Sect. 2.2 to minimize the loss function in order to train the parameter $\boldsymbol{w}$, that is:

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n} y^{(i)} \log(h(x^{(i)})) + (1 - y^{(i)}) \log(1 - h(x^{(i)})) \tag{2.21}$$

This loss function is also called cross-entropy. When the value of $y^{(i)}$ is 1 or 0, the loss function $J(x^{(i)}, y^{(i)}, \boldsymbol{w})$ on a single sample corresponds to the left or right part of formula (2.21) respectively. As shown in Fig. 2.5, when $y^{(i)} = 0$, if $h(x^{(i)})$ approaches 0, the value of the loss function becomes smaller, as too does the slope. The opposite is true if $y^{(i)} = 1$. The logistic function generates larger penalty values and larger gradients for incorrectly classified samples.

The logistic regression model defines linear binary classification problems from the perspective of regression probability. Figure 2.6a shows a graphical representation of the linear classifier. For dark-colored samples, $y = 0$; for light-colored samples, $y = 1$. The boundary in the middle is the linear classification boundary $z(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} = 0$, which is obtained through training. When $z(\boldsymbol{x}) < 0$, that is, a point is above the boundary, the predicted value is 0; otherwise, the predicted value is 1.

The linear classifier offers good interpretability, but in practice, the boundary of samples is not linear for a nonlinear classifier shown in Fig. 2.6b. This means that different parameter models need to be defined, such as the polynomial model, the SVM, and the neural network model, to learn complex nonlinear classifiers. Nevertheless, it is worth noting that the nonlinear classification problem can also be split into two parts:

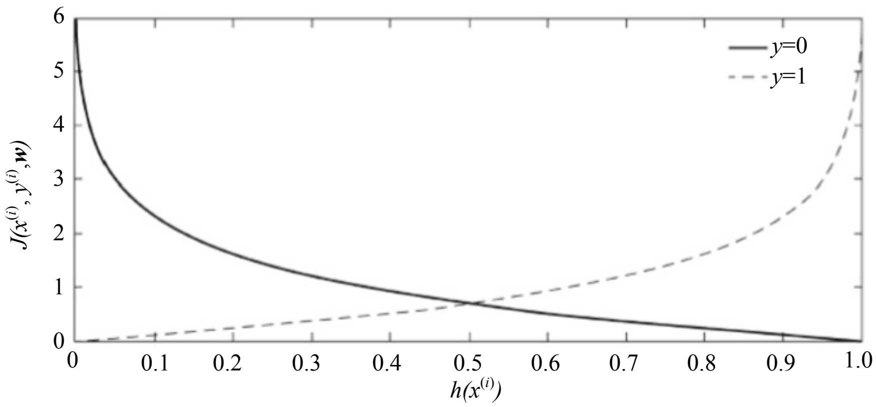


Fig. 2.5 Curve of a relationship between a single sample and a loss function

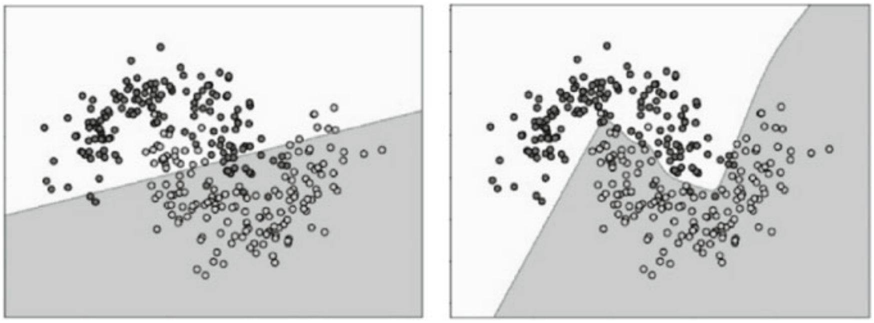


Fig. 2.6 Representation of the linear classifier and the nonlinear classifier

(1) Mapping samples to linear space by using the feature function.
(2) Learning the final classification boundary by using the linear classification model.

Chapter 3 will describe how to use the logistic regression model to train neural networks.

2.4 Overfitting and Underfitting

In machine learning, the terms overfitting and underfitting mean that the selected model does not optimally fit the data. In other words, the model is either too complex or too simple. An overfit model typically includes more parameters than the features of the data. In order to fit as much data as possible, some incorrect samples are included. However, these parameters may produce noise due to overfitting of data. The opposite of overfitting is underfitting, in which the selected parameter or model is not sufficiently complex. For example, using a linear model to fit a nonlinear structure can be considered as underfitting.

Figure 2.7 shows the problems of overfitting and underfitting in the classification problems. For a nonlinear model, if the linear model in Fig. 2.7a is used, it will not be possible to fit the nonlinear boundary. This case is called underfitting, in which the model cannot provide a good prediction effect. To fit as much data as possible, the complex model shown in the curve in Fig. 2.7c may be selected. Although such a model distinguishes the data completely, it does not optimally fit the data features. For a new test point, this overfit model may cause an error in distinguishing the data. The curve in Fig. 2.7b offers better generalization and is therefore the model of choice.

If the regression problem shown in Fig. 2.8 is fit by using a light-colored polynomial curve, although all data points can fit perfectly, the features of the data are lost. This means that the model cannot be satisfactorily generalized. For example, for

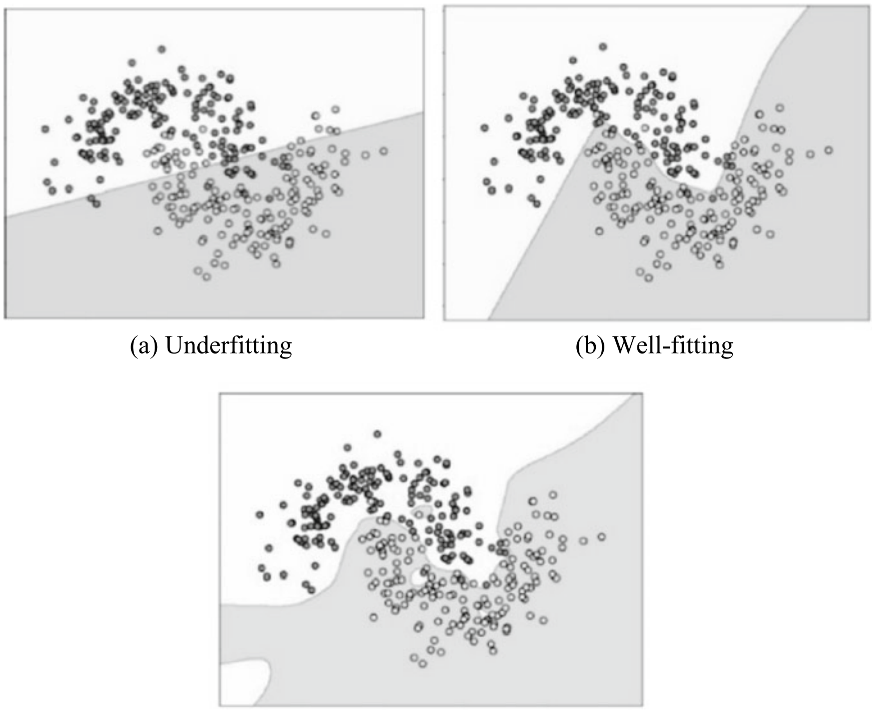
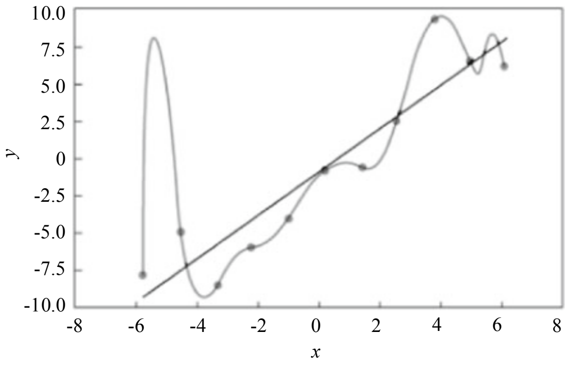


Fig. 2.7 Underfitting, well-fitting, and overfitting in classification problems

Fig. 2.8 Overfitting in regression problems



any unknown data point between the first point and the second point on the left, the predicted value is extremely large, resulting in a significant deviation. Conversely, the dark-colored linear model reflects a change trend of the data and can achieve higher accuracy when predicting a new data point.

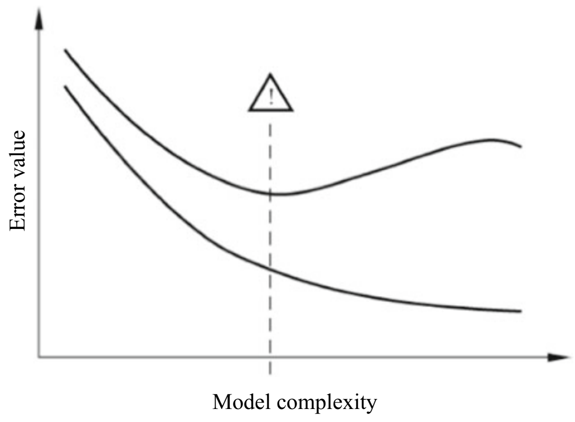


Fig. 2.9 Overfitting and underfitting of the model

In the preceding problem, the concepts of overfitting and underfitting are explained from the perspective of the graph. Now we will explain them from the perspective of data.

In a machine learning problem, the data is comprised of both training data and testing data—the model needs to be selected and trained based on the training data, and the testing data is invisible to the model. Using an underfit model would achieve a noticeably poor effect on the training data. So how can we determine whether the model is overfit? We can extract some samples from the training data to form one piece of validation data. Assuming that the distribution of validation data is similar to that of the training data and testing data, we can determine whether the model is overfit based on the model's performance on the validation data.

In Fig. 2.9, the horizontal coordinate represents model complexity, the vertical coordinate represents an error value, the lower curve represents an error on the training data, and the upper curve represents an error on the validation data. We can see that, when the model complexity increases, the error on the training data gradually decreases. This is because the complex model fits the data more easily. However, the error on the validation data decreases before it increases, because the model undergoes a gradual transition from underfitting to overfitting. The validation data is intended to help select the model at the dashed line. The model optimally fits the data, meaning that we can expect a better generalization effect—the algorithm can better adapt to new samples (testing data).

Chapter 4 will describe some methods for enhancing the generalization ability of the model to prevent overfitting. Controlling the model complexity can alleviate overfitting and help learn the rules behind the data.